Algoritmos y Estructuras de Datos II
Segundo parcial – 2do cuatrimestre 2017

A

| ? | A | - |
1 2 3

Aclaraciones

- El parcial es **a libro abierto.**
- Cada ejercicio debe entregarse **en hojas separadas.**
- Incluir en cada hoja el número de orden asignado, número de hoja, apellido y nombre.
- Al entregar el parcial, completar el resto de las columnas en la planilla.
- Cada ejercicio se calificará con **P**romocionado, **A**probado, **R**egular, o **I**nsuficiente.
- El parcial está aprobado si el primer ejercicio tiene al menos **A**, y entre los ejercicios 2 y 3 hay al menos una **A**.

Ej. 1. Diseño

La empresa proveedora de cable DirectVision quiere diseñar su sistema para administrar los clientes y los paquetes especiales que contratan. El sistema deberá permitir que los clientes se adhieran cuando deseen a un paquete especial. También se debe permitir que un cliente se dé de baja tanto de un paquete especial como del servicio de cable. Debido a la gran volatilidad en el universo televisivo se suelen agregar y eliminar paquetes especiales constantemente. Eliminar un paquete produce la baja de todos sus suscriptores.

TAD DirectVision

observadores básicos

ObtenerClientes : DirectVision $\longrightarrow$ conj(cliente)

ObtenerPaquetes : DirectVision $\longrightarrow$ conj(paquete)

ObtenerClientesPorPaquete : DirectVision × paquete p $\longrightarrow$ conj(cliente) $\{p \in ObtenerPaquetes(t)\}$

generadores

Iniciar : $\longrightarrow$ DirectVision

AgregarCliente : DirectVision t × cliente c $\longrightarrow$ DirectVision $\{c \notin ObtenerClientes(t)\}$

AgregarPaquete : DirectVision t × paquete p $\longrightarrow$ DirectVision $\{p \notin ObtenerPaquetes(t)\}$

EliminarCliente : DirectVision t × cliente c $\longrightarrow$ DirectVision $\{c \in ObtenerClientes(t)\}$

EliminarPaquete : DirectVision t × paquete p $\longrightarrow$ DirectVision $\{p \in ObtenerPaquetes(t)\}$

DarDeAltaPaquete : DirectVision t × cliente c × paquete p $\longrightarrow$ DirectVision
$\{c \in ObtenerClientes(t) \wedge p \in ObtenerPaquetes(t) \wedge c \notin ObtenerClientesPorPaquete(t,p)\}$

DarDeBajaPaquete : DirectVision t × cliente c × paquete p $\longrightarrow$ DirectVision
$\{c \in ObtenerClientes(t) \wedge p \in ObtenerPaquetes(t) \wedge c \in ObtenerClientesPorPaquete(t,p)\}$

axiomas

...

Fin TAD

Se debe realizar un diseño que cumpla con los siguientes órdenes de complejidad en el peor caso, siendo n la cantidad de clientes, m la cantidad de paquetes que ofrece la empresa y c la cantidad de paquetes de un cliente:

- AgregarCliente: $O(log\ n)$
- AgregarPaquete: $O(log\ m)$
- DarDeAltaPaquete: $O(log\ n + log\ m)$
- DarDeBajaPaquete: $O(log\ n + log\ c)$
- EliminarCliente: $O(log\ n + c)$
- ObtenerClientesPorPaquete: $O(log\ m)$

1. Escriba la estructura de representación del módulo DirectVision explicando detalladamente qué información se guarda en cada parte de la misma y las relaciones entre las partes. Describa también las estructuras de datos subyacentes. Tenga en cuenta que los paquetes y clientes poseen una relación de orden y se pueden comparar en $O(1)$.
2. Escriba el algoritmo para darse de baja de un paquete y justifique el cumplimiento de los órdenes solicitados. Para cada una de las demás funciones, descríbalas en castellano, justificando por qué se cumple el orden de complejidad pedido.

Ej. 2. Ordenamiento

Representamos un intervalo de días dentro de un año como un par de enteros $[a, b]$, con $a < b$ y ambos entre 1 y 365. Dado un intervalo $[a, b]$, decimos que otro intervalo $[c, d]$ *interrumpe* al primero, si $a < c < b$ o bien $a < d < b$, es decir, si el segundo intervalo empieza y/o termina dentro del intervalo $[a, b]$. Notar que un arreglo puede interrumpir hasta dos veces a otro (i.e., cuando está contenido en éste).

Dado un arreglo de intervalos, el *nivel de conflicto* de un intervalo es la cantidad de veces que es interrumpido por otros intervalos del arreglo.

Dar un algoritmo que ordene los intervalos de un arreglo en forma creciente según sus niveles de conflicto, con una complejidad temporal de $O(n)$, siendo n la cantidad de intervalos del arreglo.

La función a implementar debe recibir un arreglo de *intervalo*, donde *intervalo* se representa con tupla $< ini : nat, fin : nat >$.

Ej. 3. Dividir y Conquistar

Un arreglo de enteros se dice CO (*concatenación de dos ordenados*) si es la concatenación de dos arreglos ordenados cada uno de ambos en forma creciente (donde alguno o ambos podrían ser vacíos).

Ejemplos que son CO: $[10, 20, 30, 1, 2], [1, 2, 1], [10, 20, 20], [3, 2, 2, 2], [2, 1], [1], []$. Ejemplos que no lo son: $[3, 2, 1], [3, 1, 2, 3, 4, 1, 2], [4, 5, 4, 5, 4, 5], [6, 2, 3, 2]$.

Usando la técnica de Dividir y Conquistar, escribir un algoritmo que, dado un arreglo de $n = 2^k$ elementos enteros, determine la longitud del sub arreglo contiguo más largo posible que sea CO, con complejidad temporal estrictamente menor que $O(n^2)$ en el peor caso.

Por ejemplo, para $[4, 1, 2, 1, 1, 2, 3, 4, 5, 6, 2, 1, 1, 2, 2, 2]$ la respuesta es 9 (por $[1, 2, 1, 1, 2, 3, 4, 5, 6]$).

Se pide

1. el algoritmo en pseudocódigo
2. justificar su corrección, indicando claramente las fases de la técnica mencionada
3. justificar su complejidad temporal.

Ej 1) 1.

DirectVisión se ~~explica~~ representa con estr donde estr es tupla de
< Clientes : DicAVL (cliente, DicAVL (Paquete, PackStr)),
Paquetes : DicAVL (Paquete, ConjLineal (cliente)) >
donde PackStr es tupla de
< itPaquete : itDicAVL,
itCliPaquete : itConjLineal >

Estructuras usadas:

DicAVL : es un diccionario basado en un AVL donde la clave (en estos casos un Nat) es la que se compara. El mismo, por ser AVL, tiene poder de inserción en O(Log(n)) donde n es la cantidad de elementos que tiene. (lo mismo para obtener y borrar)

itDicAVL : es un iterador del DicAVL y puede acceder a la definición del elemento en O(1), sea para leerla o cambiarla.

ConjLineal : es un conjunto basado en nodos encadenados donde para ver si existe un elemento o borrarlo SIN iterador es O(#elementos) pero con un iterador se puede borrar en O(1) ya que está basado en Nodos encadenados.

itConjLineal : es un iterador del ConjLineal, permite borrar el elemento del conjunto en O(1).

[Casi me olvido de explicar la estructura, la explicación está en la parte de atrás de la 2da hoja.]

ok

2.

No hace falta, comenta el

DarDeBajaPaquete (in/out e: estr, in c: cliente, in p: Paquete)

1: PaqCliente ← &Obtener(e.clientes, c) $O(\log n)$

2: PaqYStr ← Obtener(PaqCliente, p) $O(\log c)$

3: Borrar(PaqYStr.itPaquete, PaqYStr.itCliPaquete) $O(1)$

4: Borrar(PaqCliente, p) $O(\log c)$

5: ~~Definir(e.clientes, PaqCliente)~~ // Por si no puedo usar & $O(\log n)$

↳ Si no pudieses tendrías que devolver copia y NO CIERRA NADA!

Complejidad: $O(2\log n + 2\log c) = 2.O(\log n + \log c)$

Explicación:

1. Obtengo el diccionario de paquetes del cliente en $\log(n)$ (por cant. de clientes)
2. Obtengo el paquete struct en $\log(c)$ (por todos los paq. del cliente)
3. Con el iterador del Conj Lineal y el Conj Lineal borro en $O(1)$ al cliente del paquete.
4. Borro el paquete en el cliente. Por complejidad dicho.
5.) Por si las dudas (y porque puedo hacerlo) vuelvo a pisar los paquetes del cliente en el cliente.

— o —

Agregar Cliente:

Como los clientes en estr son un DicAVL, agrego al cliente en $O(\log n)$ al dicAVL, junto con la creación de un dicAVL de paquetes para ese cliente (en $O(1)$) como definición.

Agregar Paquete:

Primero creo un Conj Lineal de clientes vacío ($O(1)$) y luego con el paquete como clave lo agrego al DicAVL de paquetes (en estr) con la complejidad de $O(\log m)$ por ser AVL.

NOTA

aclarar que se puede
por que no existe...
(o si también...)

Dar de AltaPaquete:

Primero obtengo el ConjLineal de clientes en el Paquete ($O(\log m)$), Agrego al cliente ($O(1)$) y me quedo con los iteradores

Luego creo un PaCliStr y guardo los iteradores del Conj lineal y de Paquete ($O(1)$).

Después obtengo los Paquetes del cliente en Clientes ($O(\log n)$)

Agrego al PaCliStr recién creado con la clave Paquete ($O(\log C) \leq O(\log m)$, ya que el cliente puede tener o no todos los Paquetes que existan) $\Rightarrow O(\log n + \log m)$

Eliminar Cliente:

Es muy parecido a dar de baja Paquete pero en vez de buscar un Paquete determinado, recorro todos los paquetes del DicAVL en el cliente y después, con los iteradores al Conj Lineal puedo borrar en $O(1)$ al cliente del Paquete.

Entonces, obtener el DicAVL del cliente es $O(\log n)$ y recorrer todos los paquetes para borrarlos es $O(C)$, porque borrar el Paquete es $O(1)$. Dando como resultado: $O(\log n + C)$

Detalle: Si recorrer el DicAVL no es $O(C)$ (recorrerlo, no me importa el orden) me creo un ConjLineal de Paquetes en el cliente y guardo el iterador del Conjunto a PaCliStr, para borrarlo en $O(1)$. Y en el ConjLineal guardo el itDicAVL para recorrerlo y acceder al PaCliStr.

(DicAVL podría generar una lista InOrder y recorrer esa lista, todo en complejidad $O(C)$) (o tener un Conj interno para recorrerlo EN forma desordenada)

ObtenerClientesPorPaquete:
Fácil, obtengo el ConjLineal de Clientes en Paquetes de estr.
Complejidad: O(Log m).

EliminarPaquete:
obtengo los ClientesPorPaquetes y para cada paquete borro primero el paquete en el cliente, y luego borro el paquete en paquetes.
Podés usar DarDeBajaPaquete.

Explicación de la estructura:

Paquetes: Aquí tengo un diccionario de paquetes asociados a una Conj. de clientes, que después en cada cliente se agregará el paquete.

Qué clientes guarda para cada paquete?

Clientes: Tengo un dicAVL para que, dado un cliente, obtenga los paquetes en O(Log n). Después tengo un DicAVL para estos paquetes del cliente donde la clave es el paquete que el cliente tiene y la definición es un struct asociado al cliente en estr. Paquetes. Así se puede borrar el cliente en paquetes en O(1).

Cuáles? Todos?

Ej 2)

Ejemplo:

Intervalos = [[2,3], [1,10], [5,9], [3,4], [8,12], [2,3]]

Indintervalo = 0, 1, 2, 3, 4, 5

Dias = [, [1], [0,5], [0,3,5], [3], [2], [4], [2], [1], [4], [ ]]

Pos. = 0, 1, 2, 3, 4, 5 ... 8, 9, 10 ... 12 ... 365

Conflictos = [[0,3,5], [2], [4], ..., [1], ... [ ]]

0, 1, 2, ..., 9, ... 12

#Conflictos ≤ 2·#Intervalos

Conf. por. ord = [0, 1, 2, 9]

Indices interv. ordenados = [0, 3, 5, 2, 4, 9]

Explicación:

Primero me genero un array (365) de listas enlazadas. (Dias) de nada?

Recorro los intervalos y le asigno el índice del intervalo al a los días que corresponden en el array de Dias. Ej [2,3] => Dias[2], Dias[3]. (Como es una lista enlazada puedo agregar el índice del intervalo en O(1)).

Luego recorro los intervalos de vuelta y me fijo cuántos conflictos hay en Dias de ese intervalo, los conflictos los calculo sumando la longitud de las listas. cuáles?

Ej: [5,9] => long(Dias[7]) + long(Dias[6]) + long(Dias[8])

y da como complejidad O(n*365) => O(n) Por qué?

NOTA

→ Nunca lo mencionas!!

Me olvidé de aclarar que Conflictos es un array [2.#Intervalos] de Lista Enlazada ya que no puede haber más del doble de intervalos como conflictos, dando como Complejidad: $O(2n) \Rightarrow O(n)$

Después solo queda recorrer todos los conflictos y por las listas enlazadas que no esten vacías, meto el índice en otra lista enlazada de ConflictosPorOrdenar. $O(2n) \Rightarrow O(n)$ ¿cual?

Ahora solo queda recorrer esta lista de ConflictosPorOrdenar y tengo el índice de los intervalos ordenados. los meto en una nueva lista (que es la de IntervalosOrdenados y listo).

Complejidad: $O(n + n*365 + 365 + 2n + 2n + n) \Rightarrow O(n*365) \Rightarrow O(n)$.

Se habla del array conflictos pero no se explica que guarda.

FALTA ALGO DE CLARIDAD EN LA EXPLICACIÓN...

```
Ordenar-Intervalos (In intervalos: Arreglo(intervalo) )
Dias ← Arreglo<listaEnlazada>(365)                                  O(365)
for i To Long(intervalos) do                                        O(n)
    Agregar-Atras(Dias[intervalos[i].ini], i)                       O(1)
    Agregar-Atras(Dias[intervalos[i].fin], i)                       O(1)
end
//Ahora calculo los Conflictos
Conflictos ← Arreglo<listaEnlazada>(2*Long(intervalos))             O(2*n)
for i To Long(intervalos) do                                        O(n*365)
    ini ← intervalos[i].ini +1                                      O(1)
    fin ← intervalos[i].fin ; CantConfl ← 0                         O(1)
    while ini < fin do                                              O(365)
        CantConfl ← CantConfl + Long(Dias[ini])                     O(1)
    end
    Agregar-Atras(Conflictos[CantConfl], i)                         O(1)
end
// Ahora Tengo un array indexado por Cant. de conflictos con los intervalos
// que Tienen esa cantidad de conflictos.
ConfParaOrdenar ← ListaEnlazada()                                   O(1)
for i To Long(Conflictos) do                                        O(2*n)
    If( Long(Conflictos[i]) > 0 )                                   O(1)
        AgregarAtras(ConfParaOrdenar, i)                            O(1)
    end
end
```

Este paso no es necesario. Se puede acá mismo ir reconstruyendo el orden (tal como se hace en la página siguiente)

No! Acá i es la cantidad de conflictos (no) es lo que se quiere ordenar (!)

ah... ya entendí

NOTA

```
ret ← Arreglo<Intervalo>(Long(Intervalos))            O(n)
itOrd ← CrearIT(ConflictosOrdenados) ; i ← 0            O(1)
while haySiguiente(itOrd) do                            O(#IntervConConflictoN + NConfl)
    IndexConflicto ← Siguiente(itOrd)                   O(1)
    itConflicto ← CrearIT(Conflictos[IndexConflicto])   O(1)
    while haySiguiente(itConflicto) do                  O(#IntervConConflictoN)
        IndxInter ← Siguiente(itConflicto)              O(1)
        ret[i] ← Intervalos[IndxInter]                  O(copy(Intervalo))
        ++i                                             O(1)   "O(1)
        Avanzar(itConflicto)                            O(1)
    end
    Avanzar(itOrd)                                      O(1)
end
return ret                                              O(1)
```

Debería ordenar in place, pero vale.

Complejidad:

¿Qué son?

$O(\#IntervaloConConflictoN + NConf) = O(n+1)$ dado que recorremos los n intervalos que tienen o no conflictos. Si miramos los casos bordes (que todos tengan 0 conflictos) entonces el ciclo de afuera será de 1 y el de adentro de n. Y si todos tienen conflictos diferentes entonces el de afuera será de n y el de adentro de 1. $\Rightarrow O(n+1)$

No es una buena justificación

Resolución:

$O(365 + n + 2n + 365n + 2n + n+1) \leq O(370n+1)$

$\Rightarrow \underbrace{370\ O(n) + 1}_{\text{las ctes salen..}} \Rightarrow O(n)$

NOTA